Universidad
Carlos III de Madrid
www.uc3m.es

Grado en Ingeniería en Tecnologías de Telecomunicación

2016-2017

Trabajo Fin de Grado

"IMPLEMENTACIÓN GENÉRICA DEL FILTRO DE KALMAN UKF PARA MODELOS DESARROLLADOS CON MODELICA"

Víctor Merino Ramos

Director:

Luis José Yebra Muñoz (CIEMAT)

Tutor:

Jesús Cid Sueiro (UC3M)

Universidad
Carlos III de Madrid
www.uc3m.es

20 de febrero de 2017, Leganés

Contenido:

0. ABSTRACT

The evolution of the simulations at the industry, has given way to the implementation of algorithms in these. The objective of this project has been to create one of them, the Kalman filter, in a generic way so that it is compatible for the models developed in Modelica.

This project presents a state estimation of a system, for Functional Mock-up Interface (FMI) standard models. It will be developed in Modelica simulation language and will be done through the Unscented Kalman Filter. With the Dymola simulation tool, a mathematical model has been generated that represents the fall of a body towards the earth. To this model, noise has been added in the measurements in order to compare the efficiency of the estimation. A noise that has been generated by a random sequence.

Once the algorithm has been designed, it will be simulated together with the model to be able to correct the perturbations caused by the noise. This will be able to estimate the next state of the model, that is to say, to be able to predict the future of the system.

1. INTRODUCCIÓN

1.1 Resumen

El alcance de este proyecto es la estimación de estado de un sistema determinado.
Se puede entender como sistema en [2] como "una combinación de componentes que actúan juntos y realizan un objetivo determinado. Un sistema no está necesariamente limitado a los sistemas físicos, el concepto de sistema se puede aplicar a fenómenos abstractos y dinámicos[1]". Hay numerosos sistemas a nuestro alrededor, biológicos, económicos, estructurales, funcionales, etc. Nosotros nos centraremos en los sistemas dinámicos, que entendemos por aquellos sistemas que evolucionan con el tiempo. La representación de estos sistemas en tiempo discreto viene dada por la siguiente ecuación:

$$x_{k+1} = f(x_k, t)$$

Esto significa que cada estado se obtiene de una función que depende del estado anterior y del tiempo. La parte de la ciencia que estudia estos sistemas se denomina Teoría de Control.
La teoría de control se puede resumir, de forma general, en la siguiente figura:

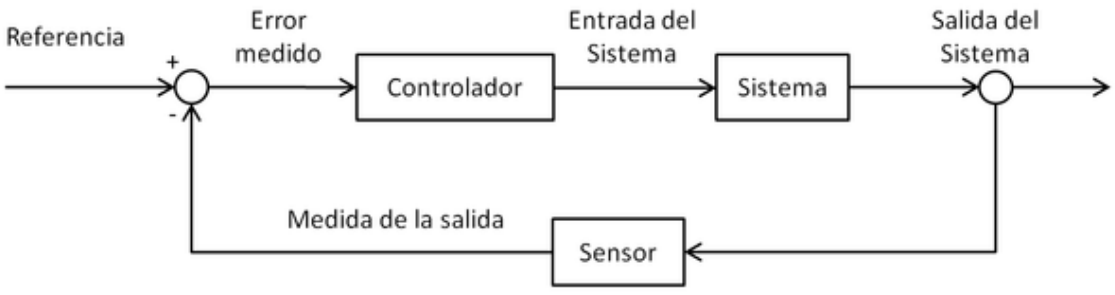


Figura 1. https://es.wikipedia.org/wiki/Teor%C3%ADa_del_control

Las principales ventajas de un sistema controlado a uno frente a uno que no lo está, es que vuelve al sistema menos sensible a las perturbaciones externas y a algunas variaciones internas respecto al comportamiento deseado. En la actualidad es posible implementar un control utilizando herramientas poco precisas y de bajo coste.

[1] Ingeniería de Control Moderna Ogata [2].

Vemos qué a la entrada del sistema, se le puede añadir una realimentación que puede ser positiva (efecto "bola de nieve") o negativa (estabilizador). A veces la realimentación es la propia salida del sistema, aunque no tiene por qué. El objetivo de este control, es converger la salida hacia una referencia deseada o intentar eliminar el error cometido en el seguimiento de dicha referencia. Un error que muchas veces está producido por el ruido que afecta a los sensores o a alguna parte del sistema entre la medición y el análisis de los datos. La siguiente figura muestra un ejemplo de como una señal puede ser perturbada por el ruido:

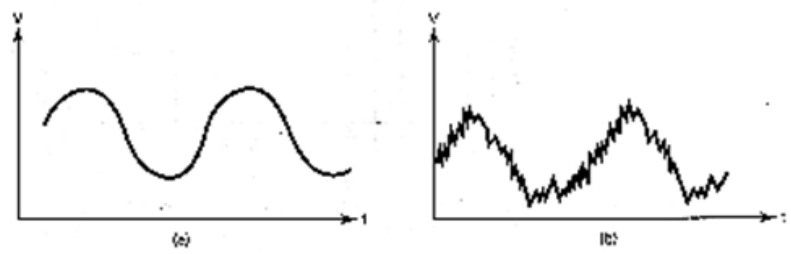


Figura 2. http://mecanicaelectric.blogspot.com.es/2013/05/ruido-electrico.html

Las perturbaciones son señales que pueden afectar negativamente a la salida de un sistema. Dependiendo de dónde se generen, pueden ser internas o externas al sistema. En los modelos ideales no existen tales perturbaciones, pero en la vida real, estas tienen un gran impacto en los sistemas sobre todo, en los que son a nivel industrial o de gran complejidad y precisión.

El interés por la estimación es muy antiguo y ha sido objetivo de muchos pensadores a lo largo de la historia. En la edad moderna, Newton desarrolló la Teoría de la Mecánica y la Gravedad Universal, lo que dio paso al desarrollo de modelos dinámicos. Un siglo después fue Gauss quién determinó la órbita de los cuerpos celestes utilizando el método de "Mínimos Cuadrados". Además de eso, años después desarrolló una variante recursiva de este método que permitía corregir un estimador previo tras una nueva observación. A partir de ese momento, el método diseñado por Gauss se convirtió en la base de numerosas teorías y técnicas de estimación. Es ya a mediados del siglo XX, cuando se comienza a aplicar en el campo de los procesos estocásticos. En ese periodo de tiempo, se dan importantes desarrollos como lo pueden ser la cibernética desarrollada

por Norbert Wiener y Arturo Rosenblueth en 1942, o la complejidad de Kolmogórov y otros muchos de pensadores como Zadeh, Ragazzini o el antecesor de Kalman, Peter Swerling y su importancia durante los años 50, en el desarrollo de radares y trayectorias de misiles.

En plena guerra fría, durante la llamada era espacial, la teoría de control cobró muchísima importancia. Fue entonces cuando Kalman, en la década de los 60, desarrollo su algoritmo basado en lo que se conoce como el modelo de espacio de estados. Un algoritmo que funciona como estimador de mínimos cuadrados en sistemas lineales discretos gaussianos y en el que nos extenderemos más adelante, ya que en él, se basa este proyecto.

1.2 Motivación

A finales del siglo XX el uso de las computadoras se generaliza tanto en la industria, como en casi todos los ámbitos de la vida cotidiana. Esto, entre una infinidad de utilidades, ha permitido la simulación de sistemas. Ya no es necesario desarrollar un modelo físico para comprobar ciertas magnitudes o para encontrar la forma más adecuada para desarrollar algo, ahora se puede simular cualquier sistema en un ordenador a un coste muy bajo y obtener los datos deseados sin tener que esperar un largo tiempo, medir algo no muy accesible y otras muchas complicaciones que se presentan a veces.

Hay numerosas herramientas para simular modelos:

- Hysys: Software para resolver problemas relacionados con procesos químicos. Sirve para simular plantas petroquímicas.
- AspenPlus: Procesos de diseño conceptual, optimización y monitorización para la industria química, polímeros y metales.
- Chemcad: Permite la simulación de destilaciones dinámicas, redes de tuberías, reactores…
- ProModel: Sirve para la simulación de procesos logísticos, manejo de materiales, talleres, grúas, bandas de transporte y mucho más.

Estas son sólo algunas de las muchas herramientas que hay para la simulación de sistemas. Existen otras muchas para desarrollos matemáticos como Matlab, que se alejan de los diseños para simular igualmente pero desde un punto más cercano a la programación.

Hoy en día, los sistemas físicos son a menudo complejos y abarcan múltiples dominios.
En nuestro caso, hemos visto un gran potencial en el lenguaje de modelado, Modelica, ya que este posee las siguientes características:

1. Componentes básicos como los enteros, reales, booleans y strings.
2. Jerarquía en los modelos, es decir, modelos descritos como submodelos conectados.
3. Tipos vectoriales y matriciales.
4. Ecuaciones y algoritmos.
5. Conexiones.
6. Funciones.

Otra ventaja es que es compatible con el lenguaje "c" de programación. Cualquier función en Modelica puede llevar código "c" integrado.
Todo esto, junto con el hecho de que este lenguaje tiene librerías para desarrollar modelos de fluidos, mecánicos, eléctricos, magnéticos, matemáticos, etc. le convierte, a día de hoy, en un lenguaje para la simulación de modelos de los más utilizados en la industria.

Este gran abanico de sectores en los que Modelica puede trabajar, hace de él un lenguaje muy útil para simulaciones complejas o en las que hay sistemas de diferentes tipos. Además de esto, se ha desarrollado recientemente, un estándar para la interconexión de simuladores de modelos desarrollados en Modelica y otros lenguajes, denominado FMI. Esto permite que se trabajen modelos diferentes en distintas herramientas, de la misma forma ya que tienen la misma estructura.

Esta necesidad, surgida en actividades de investigación en el CIEMAT (Plataforma Solar de Almería), ha sido la motivación que ha generado todo el desarrollo de este proyecto. La implementación en Modelica del filtro de Kalman para que, una vez

desarrollado, pueda ser aplicador de forma genérica a un modelo ya sea eléctrico, mecánico, termodinámico o cualquier otra disciplina.

1.3 Descripción

El alcance de este trabajo por tanto es la implementación, para modelos estándar FMI, del filtro de Kalman para sistemas no lineales desarrollados en Modelica. Dicho algoritmo se ha implementado de un modo genérico para que pueda adaptarse a cualquier tipo de modelo en dicho lenguaje.

Una vez desarrollado el algoritmo, implementaremos también un modelo FMI al que le se lo aplicaremos. Con esto se pretende corregir las perturbaciones ocasionadas por el ruido en las medidas obtenidas del modelo matemático que representa la planta real. Así podremos predecir el estado del sistema mejorando así el funcionamiento de este.

Para comprobar que el algoritmo funciona correctamente, compararemos los datos de la simulación del modelo con y sin el filtro aplicado. El objetivo final será obtener una estimación de los estados del sistema no lineal, en instantes de tiempo correspondientes al periodo de muestreo.

1.4 Estructura del proyecto

La estructura que lleva este documento es semejante a la seguida en la realización del proyecto:

Se comienza con una introducción al filtro de Kalman, más concretamente a su variante el UKF, es decir el algoritmo para la estimación de estados no lineales. Una vez comprendido el funcionamiento de dicho algoritmo, se procede a la toma de contacto con la herramienta con la que se implementará, en este caso Dymola.

En este programa aprenderemos el diseño y la implementación de modelos tanto en diagramas de bloques como con el lenguaje de programación Modelica. Se implementará entonces el algoritmo UKF en Modelica aplicándole las restricciones de dicho lenguaje y adaptándolo a los modelos estandarizados y utilizando las primitivas para la manipulación de modelos que provee el estándar FMI.

Se desarrollará paralelamente una simulación de un sistema real. Una vez acabado el filtro, se lo aplicaremos a dicha simulación y se analizaran los datos obtenidos para posteriormente sacar conclusiones.

2. METODOLOGÍA

2.1 Kalman Filter

"The Kalman filter in its various forms is clearly established as a fundamental tool for analyzing and solving a broad class of estimation problems".

Leonard McGee and Stanley Schmidt

El filtro de Kalman fue un algoritmo desarrollado por Rudolf E. Kalman en 1960. En el verano de ese año, Stanley F. Schmidt de la NASA vio su potencial y comenzó a trabajar en lo que sería la primera implementación completa de este filtro. El algoritmo, debido a su eficacia, paso rápidamente a ser parte del programa Apollo, posteriormente de la navegación del transporte aéreo C5A y pasó a ser parte imprescindible de la estimación de trayectorias a bordo de aeronaves y sistemas de control.

El algoritmo de Kalman sirve para poder identificar el estado oculto (no medible) de un sistema dinámico lineal, aun cuando este está sometido a ruido blanco aditivo. Produce estimaciones de variables desconocidas mediante el uso de una distribución de probabilidad conjunta para cada periodo de tiempo. Su simplicidad y eficacia, su estructura recursiva y su rigidez matemática, han hecho que este algoritmo se haya convertido en una pieza básica en el mundo de la navegación espacial y en el de los coches.

El resumen de este algoritmo es el siguiente: Dadas unas condiciones iniciales, el filtro supone una estimación a priori del estado del sistema. Con esta junto con la salida del sistema, el algoritmo actualiza las medidas y crea una estimación a posteriori, es decir, una estimación de cuál será el siguiente estado del sistema. Por último, se actualizan las condiciones iniciales con el estado del filtro actual y se ejecuta de nuevo el proceso.

Este funcionamiento, se resume en la siguiente figura:

Figura 3. Funcionamiento del filtro de Kalman [11].

Se resume el comportamiento del filtro para un modelo discreto. Supondremos un sistema dado por las siguientes ecuaciones lineales:

$$x_k = F_{k-1}x_{k-1} + G_{k-1}u_{k-1} + w_{k-1} \qquad w \sim (0, Q_k)$$ Ruido blanco con matriz de covarianzas Q_k

$$y_k = H_k x_k + v_k \qquad v \sim (0, R_k)$$ Ruido blanco con matriz de covarianzas R_k

Estas dos ecuaciones, "x_k" e "y_k", representan los estados y las salidas del sistema respectivamente.

Vemos que el estado tiene una variación según una matriz "F" que multiplica el estado anterior, una función G que multiplica la entrada al sistema ("u") y un ruido aditivo "w". Por otro lado, vemos que la salida del sistema varía en función del estado del sistema y de otro ruido aditivo "v".

Tanto el ruido "w" como "v" los definimos como ruidos blancos. El ruido blanco, es una señal aleatoria (proceso estocástico) que se caracteriza por el hecho de que dos de sus valores en tiempos diferentes no guardan correlación estadística, es decir, que cada valor en un instante de tiempo es pseudoaleatorio y no depende ni influye en otro valor de un instante de tiempo diferente. La siguiente figura representa el ruido generado en este proyecto.

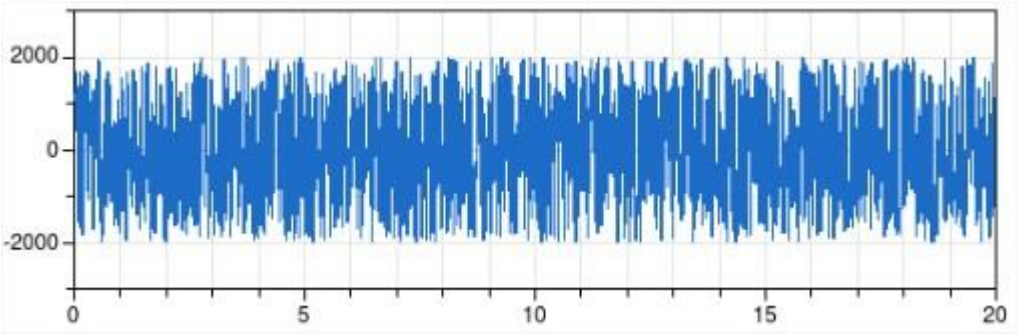


Figura 4. Ruido implementado a partir de código "c" en Modelica.

Una vez explicado el ruido, volvemos a la descripción del filtro en el supuesto modelo discreto. Conociendo las variables del sistema y la salida de este en un instante de tiempo, el filtro de Kalman intenta predecir cuál debería de ser el estado del sistema en ese momento. Hace una estimación a posteriori para intentar eliminar el error producido por el ruido:

$$\hat{x}_k^+ = E[x_k | y_1, y_2, \dots, y_k]$$

Posteriormente, conociendo las medidas de la salida y los parámetros del sistema, se puede realizar una estimación de estado, también llamada estimación a priori:

$$\hat{x}_k^- = E[x_k | y_1, y_2, \dots, y_{k-1}]$$

Notamos entonces que ambas estimaciones, "$\hat{x}_k^-$" y "$\hat{x}_k^+$", intentan predecir el estado del sistema. Pero mientras que "$\hat{x}_k^-$" hace una predicción a partir del estado actual y la salida en el instante anterior, "$\hat{x}_k^+$" utiliza también la salida de ese instante obteniendo así una mejor estimación. A continuación vemos de forma resumida el funcionamiento del algoritmo de Kalman:

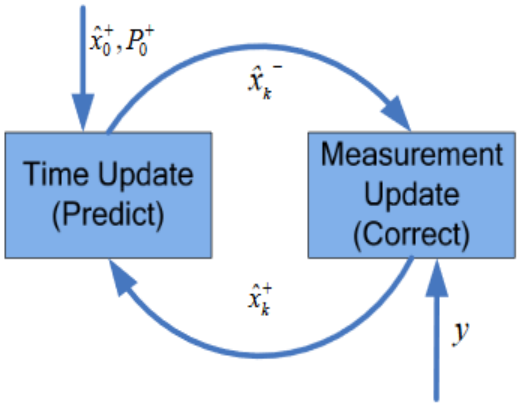


Figura 5. Principio de recursividad del filtro de Kalman [3].

Vemos que en la figura aparece la variable "P_0^+". Esta variable es la matriz de covarianzas del vector de estados inicial. También se usarán en la ejecución del filtro las matrices "P_k^+" y "P_k^-". Estas matrices, son calculadas como las varianzas estadísticas de vectores, donde la función "E()" es la media.

$$P_0^+ = E[(\hat{x}_0 - \hat{x}_0^+)(\hat{x}_0 - \hat{x}_0^+)^T]$$

Por tanto, cada iteración del filtro calcula dos estimaciones con sus respectivas matrices de covarianzas.

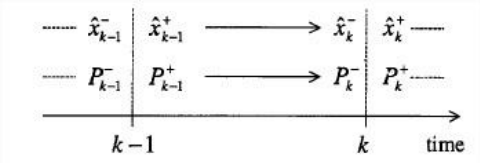


Figura 6. Estimaciones y cálculos cada instante de tiempo.

La matriz de covarianzas a priori dependerá, como hemos dicho anteriormente de la salida del instante anterior, el estado del sistema en ese momento y el ruido existente al medir este:

$$P_k^- = F_{k-1} P_{k-1}^+ F_{k-1}^T + Q_{k-1}$$

Recordemos que "F" era la función de la que variaba el estado de un sistema a su estado siguiente. Una vez conocida las covarianzas a priori, el algoritmo lo único que hace es calcular una matriz de pesos o ganancias "K_k" con la que estimará el próximo estado del sistema y su matriz de covarianzas.

$$K_k = P_k^- H_k^T R_k^{-1}$$

$$\hat{x}_k^+ = \hat{x}_k^- + K_k(y_k - H_k x_k)$$

$$P_k^+ = (I - K_k H_k)\, P_k^-$$

Representado "H" como la función de la salida del sistema en función de su estado, "R" el ruido de esta e "I" la matriz unidad.

Esta estimación discreta para sistemas lineales, sería el funcionamiento más básico del filtro de Kalman. Posteriormente se han desarrollado variantes de este algoritmo. El "Extended Kalman Filter" o EKF, fue desarrollado para ser aplicado en sistemas no lineales. EKF aproxima la distribución de estado como una variable aleatoria gaussiana y la introduce en una linealización de primer orden en un sistema no lineal. Después se desarrolló el UKF o "Unscented Kalman Filter", una nueva versión del algoritmo, parecido al anterior pero que utiliza la integración y medidas de segundo orden para la estimación de estado. UKF obtiene mejores resultados que EKF, que no converge para

todos los sistemas, aunque se cobra un coste computacional mayor. Para reducir este coste se diseñó el Square Root UKF (SR-UKF), un algoritmo idéntico al UKF pero que, en vez de utilizar la matriz de covarianzas, utiliza la raíz cuadrada de esta.

2.2 Unscented Kalman Filter "UKF"

El filtro de Kalman UKF, como he dicho anteriormente, se utiliza para la estimación de estados en sistemas no lineales. Aunque es de gran utilidad, hay que tener cuidado a la hora de introducir las variables, si se introducen medidas incorrectas el filtro puede no converger. Esto ocurre por ejemplo, cuando el ruido que perturba el sistema no tiene una función de densidad uniforme.

El algoritmo es el siguiente:

$$x_{k+1} = f(x_k, u_k, t_k) + w_k$$

$$y_k = h(x_k, t_k) + v_k$$

$$w_k \sim (0, Q_k)$$

$$v_k \sim (0, R_k)$$

En esta primera sección se nos detalla el sistema. Vemos que el estado siguiente del sistema, depende de una función que varía según el instante de tiempo, la entrada al sistema y el estado anterior, además del ruido que sufren las medidas. Vemos también que la salida del sistema varía según el estado y el instante de tiempo, además de tener su propio ruido. Por último se nos hace una descripción de los ruidos de las entradas y salidas.

$$\hat{x}_0^+ = E(x_0)$$

$$P_0^+ = E[(x_0 - \hat{x}_0^+)(x_0 - \hat{x}_0^+)^T]$$

(Ec. 2)

Aquí se realiza la inicialización de las estimaciones a posteriori, como de momento el filtro no se ha ejecutado, la estimación del estado será la estimación de los estados iniciales y la de la matriz de covarianzas dependerá del estado inicial y su media.

$$\hat{x}_{k-1}^{(i)} = \hat{x}_{k-1}^{+} + \tilde{x}^{(i)} \qquad i = 1, \dots, 2n$$

$$\tilde{x}^{(i)} = (\sqrt{nP_{k-1}^{+}})_i^T \qquad i = 1, \dots, n$$

$$\tilde{x}^{(n+i)} = -(\sqrt{nP_{k-1}^{+}})_i^T \qquad i = 1, \dots, n$$

(Ec. 3)

Para estimar el estado actual del sistema, el filtro utiliza su estimación a posteriori como muestra la figura 1. A esta estimación le suma y resta las covarianzas a posteriori del estado anterior. Estas han sido calculadas a partir de sus covarianzas a priori, que posteriormente se obtendrán en función las covarianzas de la salida y una matriz de pesos que utiliza el filtro para estimar.

En el cálculo de los puntos Sigma "$\tilde{x}^{(i)}$", el subíndice "i" representa la fila i-ésima de la matriz raíz. Posteriormente se le hará la transposición para dejarla en forma de vector vertical. Una vez como vector, "$\tilde{x}^{(i)}$", le añadimos la estimación a posteriori.

Aquí por tanto se crea un vector en el que cada elemento es un vector que representa la diferencia entre el estado y la estimación a posteriori.

$$\hat{x}_k^{(i)} = f\left(\hat{x}_{k-1}^{(i)}, u_k, t_k\right)$$

(Ec. 4)

Se calcula el estado siguiente del sistema ideal, a partir del estado anterior mediante la integración de "2n" veces la Ec. 4.

$$\hat{x}_k^{-} = \frac{1}{2n}\sum_{i=1}^{2n} \hat{x}_k^{(i)}$$

(Ec. 5)

Como muestra la figura 1, una vez tenemos el estado del sistema, el filtro realiza una estimación a priori con la hará una corrección es sus estimaciones en función de la salida. Esta estimación a priori es la media del vector de estados como muestra la Ec. 5.

$$P_k^- = \frac{1}{2n}\sum_{i=1}^{2n}\left(\hat{x}_k^{(i)} - \hat{x}_k^-\right)\left(\hat{x}_k^{(i)} - \hat{x}_k^-\right)^T + Q_{k-1}$$

(Ec. 6)

Una vez calculada la estimación a priori, se calculan sus covarianzas en función del estado, se le añaden las covarianzas del ruido que afecta a las entradas y se obtiene la matriz de covarianzas a priori.

$$\hat{x}_k^{(i)} = \hat{x}_k^- + \tilde{x}^{(i)} \qquad i = 1, \dots, 2n$$

$$\tilde{x}^{(i)} = (\sqrt{nP_k^-})_i^T \qquad i = 1, \dots, n$$

$$\tilde{x}^{(n+i)} = -(\sqrt{nP_k^-})_i^T \qquad i = 1, \dots, n$$

(Ec. 7)

Se realiza el mismo procedimiento que en la ecuación 3, salvo que esta vez se realizan las medidas con la estimación a priori en vez de a posteriori.

$$\hat{y}_k = h\left(\hat{x}_k^{(i)}, t_k\right)$$

(Ec. 8)

Obtiene los vectores de las diferentes salidas del sistema ideal, en función de los vectores de entrada.

$$\hat{y}_k = \frac{1}{2n}\sum_{i=1}^{2n}\hat{y}_k^{(i)}$$

(Ec. 9)

Calcula la media de los vectores de las salidas.

$$P_y = \frac{1}{2n}\sum_{i=1}^{2n}\left(\hat{y}_k^{(i)} - \hat{y}_k\right)\left(\hat{y}_k^{(i)} - \hat{y}_k\right)^T + R_k$$

(Ec. 10)

Genera la matriz de covarianzas en función de los vectores de salida, su media y la matriz de covarianzas del ruido que afecta a la salida.

$$P_{xy} = \frac{1}{2n}\sum_{i=1}^{2n}\left(\hat{x}_k^{(i)} - \hat{x}_k^-\right)\left(\hat{y}_k^{(i)} - \hat{y}_k\right)^T$$

(Ec. 11)

Calcula la matriz de covarianza entre las entradas y las salidas del sistema.

$$K_k = P_{xy}P_y^{-1}$$

$$\hat{x}_k^+ = \hat{x}_k^- + K_k(y_k - \hat{y}_k)$$

$$P_k^+ = P_k^- - K_k P_y K_k^T$$

(Ec. 12)

Por último, realiza la parte más importante del filtro que es la estimación del estado siguiente. Primero calcula una matriz de pesos "K_k", que varía en función de las covarianzas de los estados y la inversa de las covarianzas de salida.

A continuación, realiza una estimación del siguiente estado del sistema en función de la estimación a priori y de cómo afecta la salida al estado del sistema real. Esto lo podemos hacer gracias a "K_k", que da mayor o menor importancia a los valores en función de las covarianzas.

Por último se realiza la estimación a posteriori de la matriz de covarianzas.

2.3 Dymola como herramienta

La herramienta que se ha usado para desarrollar este proyecto ha sido Dymola. Fue creado en 1978 por Hilding Elmqvist y esta primera versión se basaba en el lenguaje de modelado dinámico, también conocido como dymola. Años más tarde, en 1996, con el fin de desarrollar un lenguaje orientado a objetos para el modelado de sistemas técnicos que sirviese para, la reutilización y el intercambio de sistemas dinámicos en formato

estandarizado, se crea el lenguaje Modelica. Por esta razón, es bastante utilizado en la industria. Con Dymola, es posible mezclar modelos de varios ámbitos de la ingeniería y exportarlos e importarlos con gran facilidad. Además de eso tiene la ventaja de que, a la hora de diseñar un sistema en Modelica, hay dos metodologías diferentes. Bien se puede crear un diagrama de bloques o elementos que se interconexionan, o bien se puede programar el modelo en el leguaje de Modelica. Ambos métodos están relacionados ya que Dymola hace de traductor entre ellos.

A parte de esto, este proyecto ha sido apoyado y supervisado por el CIEMAT. Este organismo utiliza Dymola para el desarrollo de sus modélos. Es por esto y por las numerosas ventajas que ofrece la herramienta, por lo que ha sido elegida para la implementación de este proyecto.

2.3.1 Lenguaje Modelica

La realización de experimentos a veces resulta demasiado costosa, ya sea por su dificultad, su peligrosidad, la inaccesibilidad de las variables, las perturbaciones externas que estas pueden sufrir o el tiempo de ejecución necesario a la hora de obtener resultados. Es por ello por lo que, a finales de siglo, con la aparición de la informática de alto nivel computacional a escala global, se comenzó a desarrollar software para la simulación de sistemas. El problema era, que todo lo que se desarrollaba, tenía poca transportabilidad, ya que dependía de los entornos en los que se creaba. El objetivo de Modelica fue desarrollar un lenguaje de modelado capaz de expresar la conducta de modelos, de un amplio abanico de campos de la ingeniería, y no limitarlos, a una herramienta comercial en particular.

Modelica es así un lenguaje de modelado que no tiene propietario y basa en los siguientes puntos: Encapsulación del conocimiento, interconexión topológica, modelado jerárquico, instanciación de objetos, herencia de clases y capacidad de interconexión jerarquizada. Este enfoque se adecua perfectamente a su objetivo, ya que definimos el modelo de un sistema como cualquier cosa a la que se puede aplicar un “experimento”, con el fin de responder a preguntas respecto del sistema. Por tanto, un modelo también representa un sistema, es decir, los modelos al igual que los sistemas, son por naturaleza jerárquicos.

2.3.2 FMI/FMU

Functional Mock-up Interface (FMI), define un estándar de interfaz para ser utilizado en las simulaciones por ordenador. Esta norma, FMI, proporciona los medios para el desarrollo basado en el modelado de sistemas. El proyecto que lo llevó a cabo fue MODELISAR, en 2008, y tras su finalización tres años más tarde, FMI fue gestionado y desarrollado para su implementación en Modelica.

La interfaz consiste en un pequeño grupo de funciones "C" estandarizadas para evaluar y manipular las ecuaciones del modelo, y un fichero XML que contiene la información de las variables usadas en estas. Cada variable tiene un identificador con el que es llamada por las funciones. Podemos decir por tanto que FMU consiste en el mapeo de las variables de un sistema ya que para cada uno, los identificadores son distintos y un grupo de funciones comunes a todos los modelos que permiten la obtención y utilización de estas variables.

La herramienta Dymola permite la exportación de cualquier modelo en lenguaje Modelica a uno en formato FMU, es decir, en un formato que lleva los estándares de FMI. Este avance permite, ya no solo la compatibilidad entre cualquier modelo con dicho formato, debido a sus mismas características, sino también la co-simulación de varios de estos modelos a la vez.

3. DESARROLLO

3.1 Sistema no lineal a modelar

Realizaremos la estimación de estado del ejemplo 14.2 del libro "Optimal State Estimation". La definición de este es la siguiente:

"Suponemos que estamos tratando de estimar la altitud x1, la velocidad x2 y el coeficiente balístico x3, de un cuerpo que cae hacia la Tierra. El rango de

$$\begin{aligned} \dot{x}_1 &= x_2 + w_1 \\ \dot{x}_2 &= \rho_0 \exp(-x_1/k) x_2^2 x_3/2 - g + w_2 \\ \dot{x}_3 &= w_3 \\ y(t_k) &= \sqrt{M^2 + (x_1(t_k) - a)^2} + v_k \end{aligned}$$

Como es usual, "w_i" es el ruido que afecta a la ecuación "i", y v. El ruido de medida "w_i" es la densidad del aire a nivel del mar, "k" es la constante que define la relación entre la densidad del aire y la altitud, y "g" es la aceleración de la gravedad. Los valores que utilizaremos para estas constantes, son los siguientes:

$$\begin{aligned} \rho_0 &= 2 \text{ lb-sec}^2/\text{ft}^4 \\ g &= 32.2 \text{ ft/sec}^2 \\ k &= 20,000 \text{ ft} \end{aligned} \qquad \begin{aligned} M &= 100,000 \text{ ft} \\ a &= 100,000 \text{ ft} \end{aligned}$$

Por último, describimos el estado inicial del sistema, así como una primera estimación a posteriori de la matriz de covarianzas de los estados, como:

$$x_0 = \begin{bmatrix} 300,000 & -20,000 & 0.001 \end{bmatrix}^T \qquad P_0^+ = \begin{bmatrix} 1,000,000 & 0 & 0 \\ 0 & 4,000,000 & 0 \\ 0 & 0 & 10 \end{bmatrix}$$

3.2 Implementación en Modelica

En un modelo real, a veces no se conocen ciertos parámetros o es difícil medir alguno de los estados del sistema. Por eso el filtro de Kalman funciona únicamente con la salida que produce el sistema y con la teoría de como el modelo debería comportarse. Por tanto, tal y como muestra la figura 6, del estado real solo se obtiene la salida, que introduciremos en el filtro UKF.

Figura 7. Diagrama que muestra la conexión del modelo real con el filtro.

Aparte de esta medida, el filtro internamente tiene que contener una instancia del modelo ideal, las características aproximadas del ruido que distorsiona y por supuesto, el algoritmo iterativo. En el desarrollo de este filtro, hemos realizado una instancia del modelo ideal en FMU. Dicha instancia será utilizada en el algoritmo para saber qué salida es la que debería producirse si en el modelo no hubiese ruido.

En cuanto a las características del ruido, ha sido necesario analizar previamente estas señales y, mediante métodos estadísticos, calcular los parámetros que utiliza el filtro. Esto se ha realizado mediante los bloques contador1 y contador2 representados en el anexo.

A continuación se mostrará el desarrollo seguido para la implementación del filtro y del modelo real que queremos estimar.

3.2.1 Modelo del ejemplo:

Para comprobar que el filtro funciona, se ha desarrollado en Modelica un modelo a estimar. Se trata del ejemplo 14.2 del libro "Optimal State Estimation". Se define un cuerpo que cae hacia la tierra. En dicho modelo hay tres mediciones o estados en cada instante de tiempo. Estos tres estados x1, x2 y x3 son la altitud a la que se encuentra el cuerpo, la velocidad con la que cae y el coeficiente balístico con el que atraviesa el aire respectivamente. El sistema de unidades utilizado es el Sistema Imperial. El modelo implementado es el siguiente:

```
model modelo_real
  Real x1(start=300000);
  Real x2(start=-20000);
  Real x3(start=0.001);
  parameter Real p_0 = 2;
  parameter Real g = 32.2;
  parameter Real k = 20000;
  parameter Real M = 100000;
  parameter Real a = 100000;
  output Real v;
  output Real[3,1] w;

  Modelica.Blocks.Interfaces.RealOutput y
    annotation                    (Placement(transformation(extent={{100,36},{120,56}})));
Modelica.Blocks.Interfaces.RealOutput ruido_w1
    annotation (Placement(transformation(extent={{100,-62},{120,-42}})));
  Modelica.Blocks.Interfaces.RealInput media
    annotation (Placement(transformation(extent={{-140,-20},{-100,20}})));
  Modelica.Blocks.Interfaces.RealOutput ruido_v
    annotation (Placement(transformation(extent={{100,-42},{120,-22}})));
  Modelica.Blocks.Interfaces.RealOutput ruido_w2
    annotation (Placement(transformation(extent={{100,-82},{120,-62}})));
  Modelica.Blocks.Interfaces.RealOutput ruido_w3
    annotation (Placement(transformation(extent={{100,-102},{120,-82}})));
```

Hasta aquí, la declaración de variables. Vemos que tiene las mismas que el ejemplo teórico además de: 4 variables (v y w[3]) en las que generamos ruido para después añadírselo al sistema y así pasar de un modelo teórico a uno real, y también 5 salidas que serían la salida del sistema, y los 4 ruidos para poder después calcular su varianza.
Algo muy característico de Modelica, son las coordenadas de las variables de salida. Esto sirve para la transformación entre el lenguaje de programación y la representación en diagramas.

```
algorithm
  when sample(0,0.01) then
    v:=NoiseFunctions.fun_ruido(1,100,0);
    w[1,1]:=NoiseFunctions.fun_ruido(1,4000,0);
    w[2,1]:=NoiseFunctions.fun_ruido(1,4000,0);
    w[3,1]:=NoiseFunctions.fun_ruido(1,4000,0);
    ruido_w1 :=w[1, 1];
    ruido_w2 :=w[2, 1];
    ruido_w3 :=w[3, 1];
    ruido_v :=v;
  end when;
```

Para generar el ruido que posteriormente importaremos, utilizaremos esta sección de "algorithm". Dentro de la sentencia "when", llamamos a la función sample() que se ejecuta desde el valor del primer parámetro y en intervalos del tamaño del segundo parámetro, es decir, desde el instante "0" cada 0,01 segundos los ruidos obtienen un valor pseudoaleatorio llamando a la función "fun_ruido". Gracias a la herramienta de Dymola, guardamos los ruidos como señales para analizar su varianza y así obtener una estimación de la covarianza del ruido.

Un factor a tener en cuenta es que cada vez que generamos el ruido, al ser este pseudoaleatorio, tiene una media y una varianza diferentes. Como el filtro UKF necesita conocer una aproximación de las varianzas a priori, se ha decidido generar una sola vez los cuatro ruidos y exportarlos en un fichero. Así, cada vez que se simule el modelo, se importará y utilizará el mismo ruido que le afecta. Por tanto, de cara a la declaración de variables tendremos que introducir las siguientes líneas:

```
  input Real ruido_w1;
  input Real ruido_v;
  input Real ruido_w2;
  input Real ruido_w3;
```

A partir de estas entradas de señales desde un fichero, le asignaremos estos valores a las variables que ya teníamos, sustituyendo la sección "algorithm" mostrada anteriormente por esta siguiente:

```
algorithm
    v:=ruido_v;
    w[1,1]:=ruido_w1;
```

```
    w[2,1]:=ruido_w2;
    w[3,1]:=ruido_w3;
```

Por último, en la siguiente sección vemos las ecuaciones diferenciales que describen el modelo, siendo "der()" la derivada de cualquier variable en el tiempo y "sqrt()" la función raíz cuadrada.

```
equation
    der(x1) = x2+w[1,1];
    der(x2) = p_0*exp(-x1/k)*(x2*x2)*(x3/2)-g+w[2,1];
    der(x3) = w[3,1];
    y = sqrt((M*M)+((x1-a)*(x1-a)))+v;
end modelo_real;
```

3.2.2 Algoritmo UKF:

3.2.2.1 Declaración de variables:

Para el desarrollo del algoritmo, se ha seguido las siguientes reglas de relación entre el código y las variables mostradas en la teoría:

-Toda variable que lleve un exponente "+", llevará el sufijo "_post", ya que es una estimación del estado a posteriori.

-Toda variable que lleve un exponente "-", llevará el sufijo "_priori" por la misma razón anterior.

-Toda variable que lleve un subíndice "k-1", llevará el sufijo "_ante". Si no lo lleva es porque en las ecuaciones tiene subíndice "k".

Estas son los parámetros y variables de nuestro algoritmo

```
/*----------- Parametros UKF -------------*/
parameter Integer n = 3;
parameter Integer n2 = 2*n;
parameter Real Ts = 0.01;
parameter Integer out = 1;
parameter Real
condicion_p[3,3]=[1000000,0,0;0,4000000,0;0,0,10];
parameter Real condicion_x[3,1]=[300000;-20000;0.001];
```

```
/*---------- Parametros Ruido ------------*/
parameter Real ruidov = 1.32107;
parameter Real ruidow1 = 831054000;
parameter Real ruidow2 = 11983800;
parameter Real ruidow3 = 0.0000000133032;
parameter Real cov12 = 1236750;
parameter Real cov23 = 0.00214338;
parameter Real cov13 = -0.0868243;

/*-------- Variables auxiliares ----------*/
Real contador[n,1];
Real suma[n,1];
Real contador_n[n,n];
Real suma_n[n,n];
Real contador_out[out,out];
Real suma_out[out,out];
Real contador_n_out[n,out];
Real suma_n_out[n,out];
Real vector[n,1];
Real vector2[out,1];
Real auxiliar;
Real x_inicial[n,1];
Real matriz_raiz[n,n];
Real matriz_raiz1[n,n];
Real desv_x_0[n,1];           // (x_0)-(x_post_0);
Real desv_x[n,1];             // (x_k)-(x_priori);
Real desv_y[out,1];           // (y_k)-(y_mean);

/*------------ Variables FMU -------------*/
Real[n, n] Q_ante;            // Matriz covarianzas ruido_W
Real[out, out] R_k;           // Matriz covarianzas ruido_V
Real K_k[n,out];              // Matriz de pesos
Real P_post_ante[n,n];        // P_+_k-1
Real P_priori[n,n];           // P_-_k
Real x_post_0[n,1];           // X_+_0
Real P_post_0[n,n];           // P_+_0
Real x_post_ante[n,1];        // X_+_k-1
Real x_priori[n,1];           // X_-_k
Real x_ante[n, n2];           // X_k-1
Real x_desv[n, n2];           // X~
Real y_k[out,n2];             // Y_k
Real y_mean[out,1];           // Media de Y_k
Real P_y[out,out];            // P_y
Real P_xy[n,out];             // P_xy
Real x_post[n,1];             // X_+_k
Real P_post[n,n];             // P_+_k
Real x_k[n, n2];              // X_k
```

Esta sería la declaración de las variables. A la derecha comentada, tienen la equivalencia con el algorirmo teórico descrito en la sección de metodología.

3.2.2.2 Funciones:

- fmiGetReal

Función que viene implícita con la importación de un modelo en FMU. Dicha función se utiliza para obtener el valor de una variable. Como cada variable en un FMU tiene un identificador único (sección 2.3.2), la llamada a esta función sería:

"Valor_obtenido" := fmi_functions.fmiGet_Real(fmi, {"identificador_variable"});

Donde los parámetros que se introducen son el modelo, en formato "fmi", y el identificador de la variable. La función devuelve el valor que tiene esta en ese modelo.

- fmiSetReal

Esta función modifica el valor de una variable en un determinado modelo. Para hacerlo, la llamaremos introduciendo como parámetros el modelo "fmi", el identificador de la variable y el valor al que se quiere modificar, tal y como se muestra a continuación:

fmi_functions.fmiSet_Real(fmi, {"identificador_variable"}, { "Valor_a_cambiar" });

- doStep

Al igual que las dos anteriores, esta función viene desarrollada cuando se importa un modelo FMU en modo co-simulación. Es la encargada de realizar la integración del modelo para pasar de un estado al siguiente. Para que esto suceda, hay que introducirla por parámetros el modelo a integrar, el tiempo inicial, la duración de la integración y un "boolean" que afirma o desmiente si el estado anterior ha sido valido. La función devuelve otro boolean para saber si la integración se ha realizado correctamente. Un ejemplo de llamada seria el siguiente:

"integración_bien/mal" = fmiDoStep("modelo_a_integrar", "tiempo_0", "duración", "validado");

- fFMI

La siguiente función, fFMI, es la parte más importante de todo el algoritmo. Aquí es donde se invoca a la función característica de los modelos FMU, "fmiDoStep". Esta realiza la integración del modelo para calcular así sus ecuaciones diferenciales, y poder pasar al siguiente estado del sistema. Una integración que podría desarrollarse por métodos como el de Euler o el Runge-Kutta, pero que por simplicidad se ha decidido implementarlo así, aprovechando las funciones que FMI proporciona.

Vemos que como parámetros, le introducimos el vector de estados "x" y la entrada al sistema "u". Como nuestro modelo, es decir, el ejemplo visto anteriormente en la sección 3.1, no tiene entrada alguna, hemos considerado la gravedad como entrada, aunque esta permanezca constante. De esta manera podemos realizar un filtro genérico, por si en algún momento el modelo a filtrar requiere una entrada.

También le introducimos por parámetros, el instante de tiempo en el que se encuentra el algoritmo, además del tiempo de muestreo "Ts", que representa el intervalo entre el instante "k-1" y "k".

```
function fFMI

  input fmi_Functions.fmiModel fmi;
  input Real u;
  input Real x[3];
  input Modelica.SIunits.Time Tiempo;
  input Real Ts;
  output Real x_new[3,1];

protected
  Boolean fmi_StepOK;
  parameter Real fmi_NumberOfSteps = 500
  Real fmi_CommunicationStepSize=Ts/fmi_NumberOfSteps;

algorithm
  fmi_Functions.fmiSetReal(fmi, {33554432, 33554433, 33554434, 16777232},
{x[1],x[2],x[3],u});

  fmi_StepOK := fmi_Functions.fmiDoStep(fmi,
Tiempo,fmi_CommunicationStepSize,1);

  x_new[:,1] := fmi_Functions.fmiGetReal(fmi, {33554432,33554433,33554434});
```

end fFMI;

Vemos que, dentro de la función, tenemos dos variables y un parámetro, “fmi_StepOK”, “fmi_CommunicationStepSize” y “fmi_NumberOfSteps” respectivamente. La primera variable, “fmi_StepOK”, es un tipo booleano al que le asignamos la salida de la integración, de esta forma podríamos comprobar si ha funcionado correctamente, aunque no es necesario.
El parámetro, “fmi_NumberOfSteps”, representa cuantas integraciones hará el método en un intervalo de tiempo. Por último, la variable restante, “fmi_CommunicationStepSize” será el periodo de tiempo que ocupará cada una de esas integraciones. Lo definimos como el cociente entre el tiempo de muestreo “Ts” y el número de integraciones a realizar.

El procedimiento que sigue la función se divide en tres:
Primero se modifica el estado del sistema, colocándolo así en el instante “0” de la integración. Esto se consigue con la llamada a la función “fmiSetReal”. Posteriormente se procede a realizar la integración con la llamada a “fmiDoStep”. Como se explica en dicha función, la integración se realiza entre el instante “Tiempo” y el instante “Tiempo+Ts”. Por último, se obtienen los valores de los nuevos estados y se devuelven como parámetros.

- Modelica.Math.Matrices.inv

Función presente en la librería de modélica que devuelve la inversa de una matriz.

- Modelica.Math.Matrices.cholesky

Para el cálculo de los puntos sigma, en las ecuaciones 3 y 7, es necesario calcular la raíz cuadrada de una matriz de covarianzas. Al ser una matriz, su raíz cuadrada se define de la siguiente manera:

$$\left(\sqrt{nP}\right)^{T}\sqrt{nP} = nP$$

Para obtener esta raíz cuadrada, hemos llamado a una función interna que posee Dymola en una sus las librerías de Modelica, la función “Cholesky”. Hay varias formas de invocarla, en función si la matriz introducida es triangular, pero la forma más simple es:

$$cholesky(nP) = \sqrt{nP}$$

- h_dummy

Es una función creada únicamente para obtener la salida del modelo a partir de su estado. En este caso como a la ecuación de la salida (sección 3.1) solo le afecta el estado x1, es el que introducimos por parámetros.

```
function h_dummy
  input Real estado;
  output Real salida_sig[1,1];
  parameter Real M = 100000;
  parameter Real a = 100000;

algorithm
  salida_sig[1,1] :=sqrt((M*M) + ((estado - a)*(estado - a)));
end h_dummy;
```

- return_row

Una vez obtenida la raíz cuadrada de la matriz del apartado anterior, es necesario ir sacando cada una de sus fila, tal y como se explica en la Ec. 3. Para ello, se ha desarrollado la siguiente función:

```
  function return_row
     input Real x[:,:];
     input Integer numero_fila;
     input Integer columnas;
     output Real x_new[1,columnas];
  algorithm
    x_new[1,:] := x[numero_fila,:];
  end return_row;
```

Esta función tiene como condición que la matriz introducida ha de ser cuadrada. Además como parámetro, le introduciremos el número de filas o columnas de esta matriz, así como la posición de la fila que queremos obtener.

La función obtiene todos los puntos de la fila deseada de la matriz y los introduce en un vector. Este, será la variable a devolver y que contiene los valores de dicha fila.

- rand

Para generar el ruido pseudoaleatorio, hemos recurrido a la función "rand()" del lenguaje "c" de programación. Si se llama a esta función sin introducirle parámetro alguno, te genera un número aleatorio entre 0 y la constante RAND_MAX.

3.2.2.3 Algoritmo

A continuación se describe el algoritmo en Modelica del filtro UKF:

```
algorithm

when initial() then
  /*Inicializamos Ruidos*/
  R_k[1,1]:=ruidov;
  Q_ante[1,1]:=ruidow1;
  Q_ante[2,2]:=ruidow2;
  Q_ante[3,3]:=ruidow3;
  Q_ante[1,2]:=cov12;
  Q_ante[2,1]:=cov12;
  Q_ante[3,1]:=cov13;
  Q_ante[1,3]:=cov13;
  Q_ante[2,3]:=cov23;
  Q_ante[3,2]:=cov23;
  /**********************/

  /*_____ Inicialización y Ec.2 ____*/
  x_inicial[:,1] := condicion_x[:,1];
  x_post_0 := x_inicial;
  desv_x_0 := x_inicial - x_post_0;
  P_post_0 := condicion_p;
  P_post_ante := P_post_0;
  x_post_ante := x_post_0;
  /**********************************/
end when;
```

Este when se ejecuta en el instante de tiempo "0". Inicializamos las dos matrices de covarianzas de los ruidos. Asignamos las condiciones iniciales del estado y de la matriz de covarianzas de estos.

Una vez hecho esto, entramos en un bucle que se ejecuta cada intervalo "Ts" o lo que es lo mismo, cada iteración del filtro ente "k" y "k+1".

```
when {sample(fmi_StartTime, Ts)} then
```

```
/*_________ Ec. 3 _________*/
matriz_raiz1 :=Modelica.Math.Matrices.cholesky(n*P_post_ante,true);
for j in 1:n2 loop
  if j<(n+1) then
    vector[:,:] :=transpose(return_row(matriz_raiz1,j,n));
    x_desv[:,j] := vector[:,1];
  else
    vector[:,:] := transpose(return_row(-matriz_raiz1,(j-n),n));
    x_desv[:,j] := vector[:,1];
  end if;
  x_ante[:,j] :=x_desv[:,j] + x_post_ante[:,1];
end for;
/*************************/
```

Esta parte del código calcula los puntos sigma de la Ec.3. Se calcula primero la raíz cuadrada con la función cholesky. A partir de esta se formará el vector de vectores puntos sigma, siendo la primera mitad de este la raíz transpuesta y también la segunda mitad salvo con signo negativo. Posteriormente se calcula el vector de vectores de los estados anteriores, a partir de estas desviaciones y la estimación a posteriori.

```
/*_________ Ec.4 _________*/
for i in 1:n2 loop
  vector[:,:] := fFMI(modelo_ideal.fmi,g,x_ante[:,i],time,Ts);
  x_k[:,i] := vector[:,1];
end for;
/*************************/
```

Se ejecuta un bucle para calcular el vector de vectores de estados actual. Esto se consigue integrando los estados anteriores con la función fFMI.

```
/*_________ Ec. 5 _________*/
for s in 1:n2 loop
  contador[:,1] := x_k[:,s];
  suma := suma + contador;
end for;
x_priori := (1/n2)*suma;
suma:=zeros(n,1);
/*************************/
```

Con un bucle y un contador como variable auxiliar, realizamos el sumatorio de los vectores de estado para posteriormente calcular su media. Al final, reinicializamos el sumatorio para la siguiente iteración del algoritmo.

```
/*_________ Ec. 6 _________*/
for a in 1:n2 loop
```

```
  desv_x[:,1] := x_k[:,a] - x_priori[:,1];
  contador_n := desv_x*transpose(desv_x);
  suma_n := suma_n + contador_n;
 end for;
 P_priori :=(1/n2)*suma_n + Q_ante;
 suma_n:=zeros(n,n);
 /*************************/
```

Realizamos un sumatorio mediante un bucle como hemos visto anteriormente. Este calcularemos la multiplicación de vectores por sus traspuestos y le añadiremos el ruido obteniendo asi la matriz de covarianzas de los estados, en este caso, será la matriz a priori.

```
 /*________ Ec. 7 ________*/
 matriz_raiz := Modelica.Math.Matrices.cholesky(n*P_priori,true);
 for e in 1:n2 loop
  if e<(n+1) then
vector[:,:] := transpose(return_row(matriz_raiz,e,n));
    x_desv[:,e] := vector[:,1];
  else
vector[:,:] := transpose(return_row(-matriz_raiz,(e-n),n));
    x_desv[:,e] := vector[:,1];
  end if;
  x_k[:,e] :=x_desv[:,e] + x_priori[:,1];
 end for;
 /*************************/
```

La Ec.7 sigue el mismo procedimiento que la Ec.3.

```
 /*________ Ec. 8 ________*/
 for i in 1:n2 loop
  auxiliar :=x_k[1, i];
  vector2[:,:] := h_dummy(auxiliar);
  y_k[:,i] := vector2[:,1];
 end for;
 /*************************/
```

Con un bucle obtenemos el vector de salidas a partir del vector de vectores de estados.

```
 /*________ Ec. 9 ________*/
 for w in 1:n2 loop
  contador_out[:,1] := y_k[:,w];
  suma_out[:,1] := suma_out[:,1] + contador_out[:,1];
 end for;
 y_mean := (1/(2*n))*suma_out;
 suma_out:=zeros(out,1);
 /*************************/
```

La Ec.9 sigue el mismo procedimiento que la Ec.5.

```
/*________ Ec. 10 ________*/
for e in 1:n2 loop
  desv_y[:,1] := y_k[:,e] - y_mean[:,1];
  contador_out := desv_y*transpose(desv_y);
  suma_out := suma_out + contador_out;
end for;
P_y:=(1/(2*n))*suma_out + R_k;
suma_out:=zeros(out,1);
/**************************/
```

Las Ec. 10 y 11, son muy parecidas a la Ec.6. La diferencia reside que la Ec. 10 calcula la matriz de covarianzas de la salida mientras que la Ec. 11 calcula la que existe entre las entradas y las salidas.

```
/*________ Ec. 11 ________*/
for v in 1:n2 loop
  desv_x[:,1] := x_k[:,v] - x_priori[:,1];
  desv_y[:,1] := y_k[:,v] - y_mean[:,1];
  contador_n_out := desv_x*transpose(desv_y);
  suma_n_out := suma_n_out + contador_n_out;
end for;
P_xy :=(1/(2*n))*suma_n_out;
suma_n_out:=zeros(n,out);
/**************************/
```

A continuación, se genera la estimación del sistema. Con la Ec. 12, el filtro actualiza su matriz de pesos para seguidamente utilizarla en el cálculo de la estimación a posteriori, tanto del estado como de la matriz de covarianzas.

```
/*________ Ec. 12 ________*/
K_k := P_xy* Modelica.Math.Matrices.inv(P_y);
x_post :=x_priori + K_k*(salida_real - y_mean[1,1]);
P_post :=P_priori - K_k*P_y*transpose(K_k);
/**************************/
```

Por último al filtro solo le queda reasignar las variables del estado actual a valores del estado anterior para la siguiente iteración.

```
/******* Actualización de variables *******/
x_ante :=x_k;
x_post_ante :=x_post;
P_post_ante :=P_post;
/*****************************************/
end when;
```

3.3 Adaptación del algoritmo al FMU

Como he dicho anteriormente, la base del proyecto es la implementación genérica del filtro de Kalman para modelos FMU. Este formato de modelo se divide en tres grandes partes: el código del modelo en sí, los ficheros que hacen de traductores entre Modelica y el formato FMU, como lo puede ser el archivo ".xml" que tiene asignados identificadores a las variables, y por último la parte de las funciones características de los FMU.

Esta última parte, la formada por las funciones, viene implícita en cada modelo FMU creado por Dymola en el paquete "fmi_functions". Este paquete tiene una clase principal interna llamada "fmiModel" que es el modelo interno y a partir de la cual, se hacen todas las llamadas a estas funciones.

Supongamos que queremos obtener el valor de una variable del modelo "ejemploFMU". Este modelo tendrá internamente, a parte del "fmiModel" llamado "fmi", el paquete de funciones "fmi_functions" al que llamaremos para usar la función "fmiGet_Real" y asi obtener el valor deseado. Si en vez de obtener el valor, quisiésemos modificarlo cambiando así el estado del modelo, llamaríamos a la función "fmiSet_Real". Es decir, todas las funciones observan o modifican el modelo fmi. Para hacer esto, usan los identificadores de las variables que se encuentran en el fichero ".xml".

Podríamos decir que las funciones del FMU, son todas aquellas acciones que a un modelo se le pueden ejecutar. Estas acciones tienen que llevar una descripción en la que se encuentran los identificadores de las variables, que actúan como traductores entre Modelica y el modelo.

4 PROBLEMAS

Durante la implementación de este proyecto, hemos tenido que esquivar varios obstáculos. Como la implementación y la comprobación han sido realizadas con software, todos los problemas han sido de carácter informático.

La mayoría de ellos han sido pequeños errores de al desarrollar el código en Modelica. Aparte de estos cabe destacar los tres siguientes problemas:

Otro problema que ha llevado a invertir bastante tiempo, ha sido el ejemplo utilizado descrito (sección 3.2.1). Este, en el libro, contiene algunas erratas como la gráfica del estado x1 o la pobre descripción de los ruidos. Conociendo esto, con el fin de no sacar conclusiones erróneas, la investigación se ha ceñido a los resultados obtenidos y no a los propuestos por el libro.

El más destacado de los obstáculos ha sido la generación del ruido. Es fácil generar una distribución aleatoria con cierta media, pero no lo es si se pretende que tenga una varianza determinada. Por ello, como se ha descrito anteriormente (sección 3.2.1), cuando se genera el ruido, mediante cálculos estadísticos, se calculaba la media y su varianza. Como si se genera otra vez varía al ser un proceso pseudoaleatorio, estas señales de ruido son grabadas en un fichero. A la hora de la estimación del estado, se introducen en el filtro los valores calculados de las varianzas del ruido, y se introduce este nuevamente este desde el fichero.

5 EVALUACIÓN

5.1 Resultados obtenidos

Para la verificar los resultados del filtro, primero será necesario comprobar que el modelo a filtrar es correcto. Se generará el ejemplo (sección 3.2.1.) en Modelica y se simulará con Dymola durante un periodo de tiempo para observar su comportamiento.

Como el modelo a filtrar no es real, sino que ha sido generado por software a partir de un modelo ideal y un ruido aditivo, lo primero que haremos será desarrollar un ruido con unas características determinadas para que UKF pueda hacer su estimación de estados.

Este ruido se forma a partir de una función en programación en "C". Esta es un bucle que genera con cada iteración, un punto aleatorio. Si a esta función, "fun_ruido", la llamamos como se describe en la sección 3.2.1, obtenemos el ruido tal y como muestra la figura 4. Podemos decir por tanto, que el ruido generado es pseudoaleatorio.

Para comprobarlo, comprobaremos que es una función de distribución analizando su transformada de Fourier (FFT).

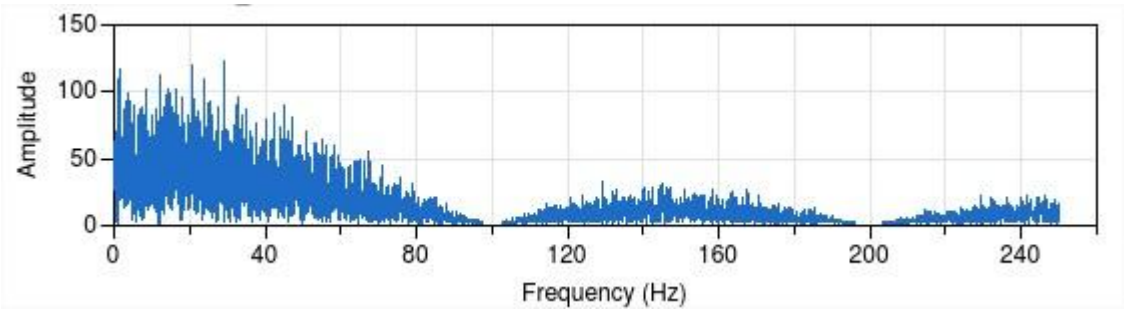


Figura 8. TFF del ruido de la figura 2 realizada por Dymola.

Podemos ver aproximadamente que la FFT de la señal del ruido tiene el aspecto de la función "sinc", mostrada en la figura 9:

$$sinc(x) = \frac{\sin(x)}{x}$$

La TFF de una señal en el dominio del tiempo, es su representación en el dominio de la frecuencia. Esto nos da una idea del comportamiento de dicha señal, tanto a lo largo del tiempo como a distintas frecuencias.

La siguiente figura muestra la representación de una "sinc" en el dominio de la frecuencia y su TFF inversa en el dominio temporal:

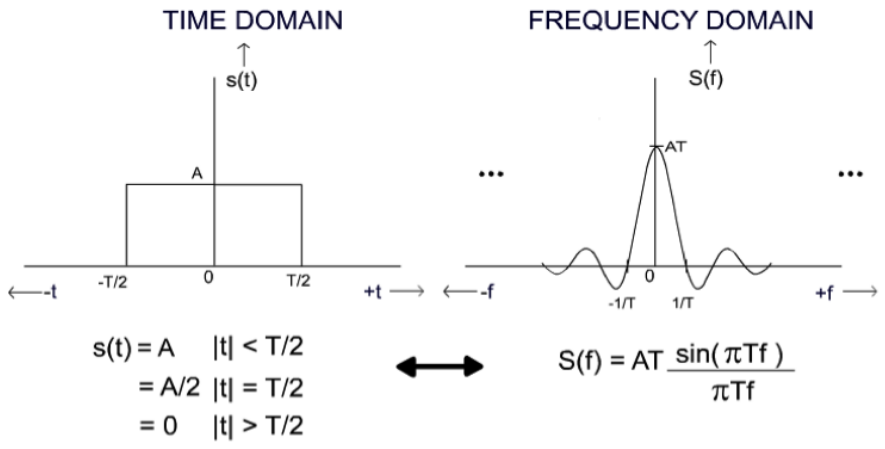


Figura 9. http://oh-jbclub.tk/wypim/fourier-transform-homework-help-fag.php

Vemos que el ruido generado posee una TFF muy parecida a un rectángulo ideal en el dominio del tiempo. Esto significa que tiene una función de densidad uniforme a lo largo del tiempo, por lo que el filtro UKF podrá realizar la estimación de estados sin problemas.

Una vez comprobado que el ruido es el adecuado, pasaremos al desarrollo del ejemplo (sección 3.2.1). Como se ha dicho anteriormente, crearemos un modelo real a partir de uno ideal añadiéndole ruido. Con el fin de mostrar todo el proceso de una forma no densa, mostraremos solo uno de los 3 estados del ejemplo, siendo el procedimiento idéntico para cada uno de ellos.

El modelo ideal será el que transformaremos a FMU para realizar la integración en cada instante de tiempo. Si por ejemplo, observamos la salida del ejemplo, esta en el modelo ideal tiene la siguiente forma:

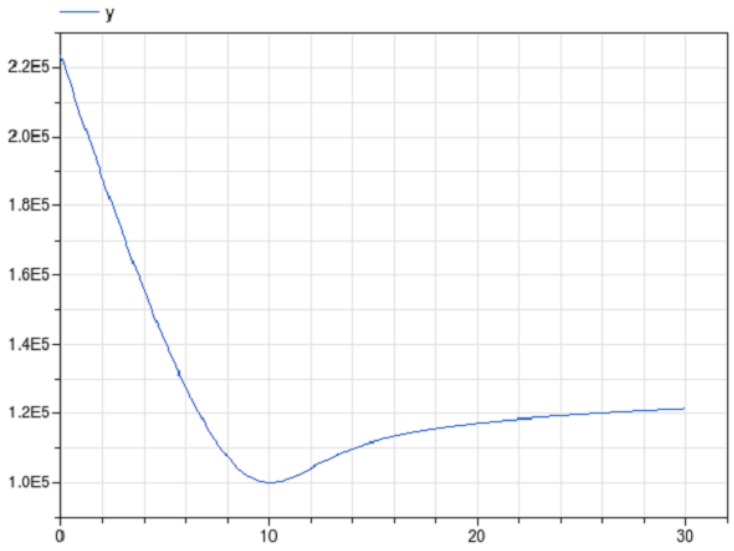


Figura 10. Evolución del estado x1 del modelo ideal a lo largo del tiempo.

El desarrollo del modelo real será idéntico al anterior, pero añadiéndole las variables "v" y "w". Estas contienen el ruido y afectan a todas las ecuaciones diferenciales, de tal forma que la salida distorsionada del sistema resulta:

Figura 11. Evolución de la salida del modelo real a lo largo del tiempo.

Se puede observar como el ruido introducido al modelo ideal, hace que el estado varíe de la figura 10 a la figura 11, distorsionando los estados y la salida del sistema.

Aplicando entonces el filtro al modelo vemos como este, desde las condiciones iniciales comienza a converger sus parámetros. En función de la salida del modelo real, y

utilizando el modelo ideal como base, el filtro va realizando estimaciones de estado a partir de estos parámetros. Vemos que con cada iteración, afina cada vez más su matriz de pesos “K_k”. La siguiente figura, muestra la convergencia del peso para el estado x1:

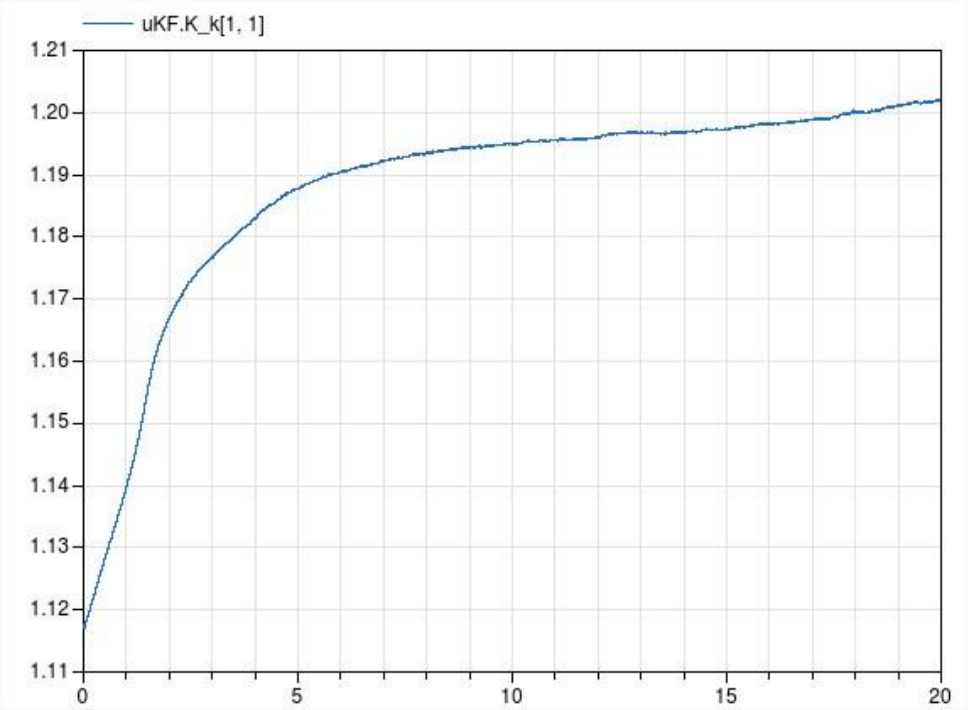


Figura 12. Evolución de la ganancia k[1].

Esta matriz de pesos es la que utiliza el filtro como base para la estimación de cada estado. Podemos ver en la figura 12, como el peso para el estado x1 converge a lo largo del tiempo. Esto significa que durante todas las estimaciones, el filtro ha conseguido desarrollar unas matrices de covarianzas que se asemejan bastante, a la realidad generada en el laboratorio. Conociendo entonces, como se afectan las señales entre sí, es decir, como afectan los ruidos a cada uno de los estados, el UKF es capaz de “predecir” o estimar el próximo estado en el que se encontrará el sistema real.

La figura 13 representa, como se asemeja la estimación realizada por el filtro, para el estado x1, al modelo real en un pequeño intervalo de tiempo.

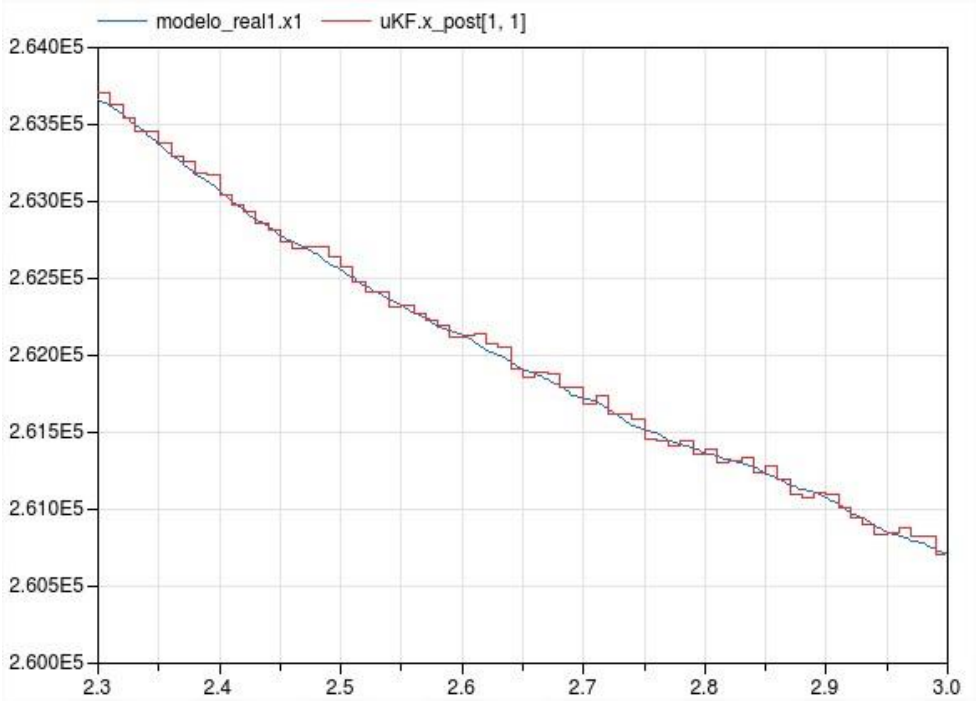


Figura 13. Representación del estado x1 del modelo real, frente a la estimación del filtro para el mismo estado.

Se puede observar como el filtro oscila alrededor del estado real. Una oscilación que representa una mayor aproximación de cómo sería el estado futuro, si en el cálculo de este no participase una distorsión producida por los valores pseudoaleatorios del filtro.

La figura 13 muestra los valores de la estimación con menor precisión de la que tiene la medición del modelo real. Una mejor aproximación de este sistema se produciría con la disminución del tiempo de muestreo "Ts", el tiempo con el que el UKF genera cada iteración.

6 CONCLUSIONES

De los resultados se deduce la eficacia del filtro realizado. Se ha probado la generación de una señal pseudoaleatoria, el desarrollo de un modelo en el lenguaje Modelica así como el algoritmo con el que se estimará y sobre todo, como representa la figura 13, el filtro funciona correctamente.

Para trabajos futuros, se podría mejorar el filtro realizando la llamada a las variables de forma automática y no por su identificador. Esto permitiría al usuario realizar las llamadas a funciones sin tener que profundizar en el formato FMU. Otro aspecto que se le podría mejorar al filtro sería el hacerle adaptativo. Si en vez de poner fija la matriz de covarianzas de los ruidos, la colocásemos como entrada para que fuese variando en función del tiempo, se obtendrían mejores resultados.

La ventaja de haberlo desarrollado en Modelica, es que al ser un lenguaje de modelado que permite herencias, este filtro puede ser encapsulado para posteriormente aplicarle mejoras y modificaciones en niveles superiores. Si a esto le añadimos que ha sido diseñado de forma genérica, en cuanto a modelos FMU se refiere, podría decirse que tiene un gran potencial en cualquier sector. Por tanto, el aspecto más importante de este trabajo es que se ha generado un filtro UKF para todos los modelos desarrollados en Modelica incluidos aquellos con formato estandarizado FMU.

Y se pueden desarrollar casi cualquier diseño FMU ya que la mayoría de los sistemas, pueden aproximarse mediante modelos aproximados. Una simulación de una máquina industrial, un cohete espacial o incluso el movimiento migratorio de un ave, podría desarrollarse en Modelica. En todos estos modelos hay un ruido que nos impide estimar su futuro, o mejor dicho, variables que afectan al sistema y que consideramos aleatorias y no podemos predecir. Lo único que hay que hacer es, analizar estos sucesos, hacer un cálculo estadístico para analizar las variaciones de este “ruido”, y ya se podría aplicar al modelo, este filtro de Kalman y por consiguiente, predecir su futuro.

7 BIBLIOGRAFÍA

[1] Simon, D. (2006). *Optimal State Estimation: Kalman, H, and Nonlinear Approaches*. http://doi.org/10.1002/0470045345

[2] Ogata, K. (2013). *Ingeniería de Control Moderna*. *Journal of Chemical Information and Modeling* (Vol. 53). http://doi.org/10.1017/CBO9781107415324.004

[3] Brembeck, J., Otter, M., & Zimmer, D. (2011). Nonlinear Observers based on the Functional Mockup Interface with Applications to Electric Vehicles. *8th International Modelica Conference*.

[4] Fritzson, P., & Thiele, B. (2016). Introduction to Object - Oriented Modeling, Simulation, Debugging and Dynamic Optimization with Modelica using OpenModelica. *Simulation*, (February 2004), 47163.

[5] Hilding Elmqvist. (1997). Modelica — A unified object-oriented language for physical systems modeling. *Simulation Practice and Theory*, *5*(6), p32. http://doi.org/10.1016/S0928-4869(97)84257-7

[6] Modelica Association. (2000). ModelicaTM - A Unified Object-Oriented Language for Physical Systems Modeling: Language Specification. *ReVision*, 1–71. http://doi.org/10.1016/S0928-4869(97)84257-7

[7] Bravo, V. A. O., Arias, M. A. Ni., & Cardenas, J. A. C. (2013). Analisis Y Aplicación Del Filtro De Kalman a Una Señal Con Ruido Aleatorio. *Scientia et Technica*, *18*(1), 267–274. Retrieved from http://200.21.217.140/index.php/revistaciencia/article/view/8241

[8] Bring, O. (2012). Introduction to Modeling and Simulation with Modelica using OpenModelica. *Simulation*, (February 2004), 47163.

[9] Åström, K. J., & Wittenmark, B. (2001). Computer-Controlled Systems: Theory and Design, (3rd edition), 555. http://doi.org/10.1007/s13398-014-0173-7.2

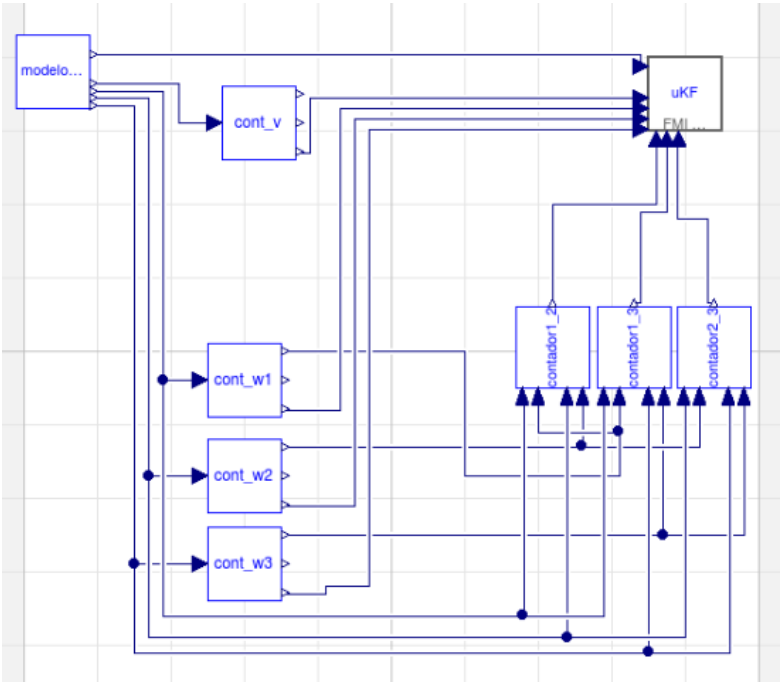
[10] http://www.cs.unc.edu/~welch/kalman/

[11] http://www.tsc.uc3m.es/~mlazaro/Docencia/Doctorado/FiltAdapt/Kalman.pdf

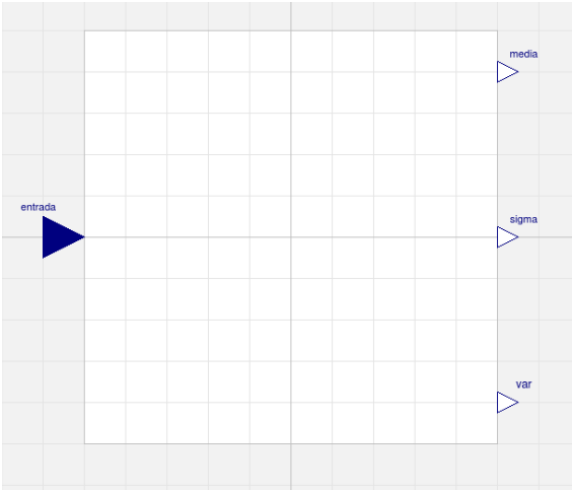
[12] http://softwaresdesimulacion.blogspot.com.es/2014_02_01_archive.html

8 ANEXO

A.1 Montaje final



A.2 Contador cálculo varianzas

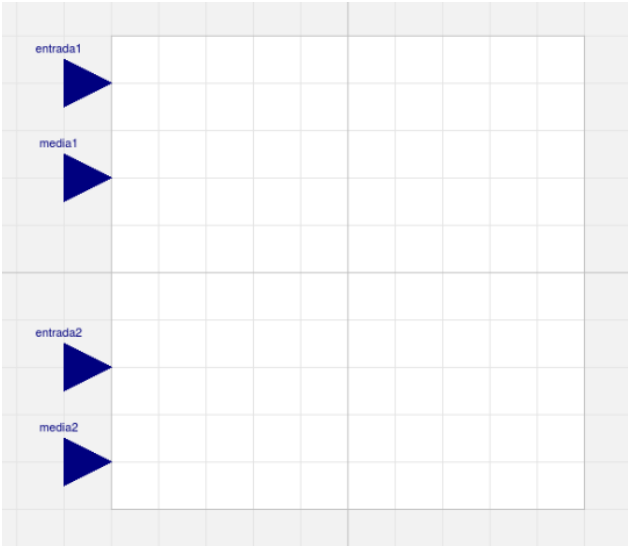


```
model contador

Real cuenta(start=0);
Real cont(start=0);
Real resta2(start=0);
Real suma_resta2(start=0);
  Modelica.Blocks.Interfaces.RealInput entrada(start=0)
    annotation (Placement(transformation(extent={{-140,-20},{-100,20}})));
  Modelica.Blocks.Interfaces.RealOutput media(start=0)
    annotation (Placement(transformation(extent={{100,70},{120,90}})));
  Modelica.Blocks.Interfaces.RealOutput var(start=0)
    annotation (Placement(transformation(extent={{100,-90},{120,-70}})));
  Modelica.Blocks.Interfaces.RealOutput sigma(start=0)
annotation (Placement(transformation(extent={{100,-10},{120,10}})));

algorithm
  when sample(0,0.01) then
            cont :=cont + 1;
            cuenta :=cuenta + entrada;
            media :=cuenta/cont;
            resta2 :=entrada - media;
            suma_resta2 :=suma_resta2 + (resta2*resta2);
            var :=suma_resta2/cont;
            sigma:=sqrt(var);
  end when;
  annotation (
    Icon(coordinateSystem(preserveAspectRatio=false)),
    Diagram(coordinateSystem(preserveAspectRatio=false)));
end contador;
```

A.3 Contador cálculo covarianzas




```
model contador2
  Real resta1(start=0);
  Real resta2(start=0);
  Real suma_restas(start=0);
  Real cont(start=0);
  Modelica.Blocks.Interfaces.RealInput entrada1
    annotation (Placement(transformation(extent={{-140,60},{-100,100}})));
  Modelica.Blocks.Interfaces.RealInput entrada2
    annotation (Placement(transformation(extent={{-140,-60},{-100,-20}})));
  Modelica.Blocks.Interfaces.RealInput media1
    annotation (Placement(transformation(extent={{-140,20},{-100,60}})));
  Modelica.Blocks.Interfaces.RealInput media2
    annotation (Placement(transformation(extent={{-140,-100},{-100,-60}})));
  Modelica.Blocks.Interfaces.RealOutput covarianza
annotation (Placement(transformation(extent={{100,-10},{120,10}})));

algorithm
  when sample(0,0.01) then
            cont := cont+1;
            resta1:=entrada1-media1;
            resta2:=entrada2-media2;
            suma_restas := suma_restas + (resta1*resta2);
            covarianza:=suma_restas/cont;
  end when;
  annotation (Icon(coordinateSystem(preserveAspectRatio=false)), Diagram(
       coordinateSystem(preserveAspectRatio=false)));
end contador2;
```